A Framework for Expressing and Estimating Arbitrary Statistical Models Using Markov Chain Monte Carlo

Todd L. Graves

August 8, 2003

Keywords: Java, Metropolis–Hastings algorithm, reversible jump, updating schemes, expressing statistical models

Abstract

YADAS is a new open source software system for statistical analysis using Markov chain Monte Carlo. It is written with the goal of being extensible enough to handle any new statistical model and versatile enough to allow experimentation with different sampling algorithms. In this paper we discuss the design of YADAS and illustrate its power through five challenging examples that feature unusual likelihood functions and/or priors, require special measures to obtain efficiently mixing algorithms, or require advanced techniques such as reversible jump.

1 Introduction

The Markov chain Monte Carlo (MCMC) revolution made Bayesian analysis of a larger class of problems feasible, but practitioners still face obstacles in analysis of their models, particularly new ones. It is often straightforward in principle to implement a MCMC analysis of a model, but in reality the process has many pitfalls that can absorb researchers' time. Many researchers, then, would welcome software that relieves them of as much as possible of the difficulties in model implementation. Berger (2000), in his JASA vignette about Bayesian analysis, argues that "creation [of 'automatic' Bayesian software] should... be a high priority for the profession." Berger lists a number of software packages, including many special purpose packages and also some useful packages of more general applicability. In the present paper we discuss our contribution to the Bayesian software effort, a system written in Java that in its development phases has been known as YADAS, an acronym for "yet another data analysis system," and pronounced as if it were a contraction for "yada yada yada." We are distributing the source code, together with a tutorial and examples, at `yadas.lanl.gov`.

Berger (2000) cites strengths of Bayesian analysis that also serve as obstacles to the creation of software packages. First, Bayesian analysis is applicable to virtually all statistical models, and the task of creating software that is so versatile or that at least is relatively easy to extend, is daunting. YADAS was built with the intention of making it extensible enough in principle to handle all (especially new) models, because the intended users are statistical modeling researchers who create novel models with potentially several levels of hierarchy and several data and prior information sources. In this paper we list some impressive successes of YADAS in fitting some highly nontrivial models that were not related to each other, and argue that the framework is extensible to other models not envisioned, in a relatively straightforward way.

The second obstacle listed by Berger (2000) is that much recent research has extended the class of useful computational techniques for Bayesian analysis, and this rapidly growing tool set might be interpreted as a threat to make today's software obsolete. Our approach is based on the "variable-at-a-time" Metropolis algorithm, but we also provide a convenient framework for users to design and use arbitrary ways of updating values of unknown parameters, particularly in cases where one wants to move multiple correlated parameters simultaneously. In fact, we intend for YADAS to be useful as an environment for research into new methods for improving mixing in MCMC.

1.1 Philosophy and design decisions of YADAS

We take a moment to describe some of the motivating ideas behind the design of YADAS. First, a well-designed system will make it easy to make small changes to analyses already performed, including changing prior parameters, but also distributional forms, link functions, and so forth. Any of these changes should be possible using one line of code or one value in an input file. Importantly, analysts should not have to perform algebraic calculations such as full conditional distributions. Three defining characteristics of the YADAS effort are the forsaking of exact full conditional distributions, an emphasis on saving human time sometimes at the expense of computational efficiency, and open source distribution.

- *Variable-at-a-time, random walk Metropolis steps.* The assumptions inherent in our software approach include the belief that most posterior distributions can be explored adequately using componentwise Metropolis steps centered on the unknown parameters' current values. Algorithms built in this way will, except in pathological cases, generate samples from the posterior distribution. See Chib and Greenberg (1995), and Tierney (1994). Metropolis *et al* (1953) and Hastings (1970) are the most important historical papers about the Metropolis algorithm. Gelfand (2000) is a useful recent review of the Gibbs sampler and related topics. An approach that by default uses Metropolis moves frees the analyst from the algebra needed to evaluate correct conditional distributions of parameters, and also from the need to choose model forms and prior distributions on the basis of their conjugacy properties. Of course, the random walk Metropolis approach requires the user to specify step sizes for Metropolis steps, but these step sizes have been easy to tune in our experience. YADAS also allows users to add Gibbs steps if they are willing to evaluate and code the conditional distributions, but we do not anticipate this being helpful.
- *Computational efficiency.* We did not emphasize computational speed in the design of YADAS; instead we concentrated on making it easy to specify models, minimizing the amount of human time necessary between creation of the model and the start of the MCMC running. YADAS's speed will not match that of special-purpose implementations, but thus far we have not found lack of speed to be an obstacle (but use of YADAS will not be practical for the largest models for which optimized code is critical). In particular, the speed of Java is underrated.
- *Open source distribution.* The fact that we are following an open source distribution model is vital for the future of this software. One of the professed strengths of YADAS is its extensibility, and this advantage would be partly destroyed by a more restrictive distribution. Certainly the target audience for the software consists of statistics researchers, especially those working on new models and algorithms. We hope that others will seize on the opportunity to develop packages built on top of YADAS. We also hope that YADAS will serve as an "enabling technology" for research, both encouraging researchers to experiment with models that might be appropriate but difficult to fit, and providing a framework for experimentation with MCMC mixing improvement techniques.

1.2 Comparison with existing systems

Several other general–purpose MCMC tools are available. First, the free, versatile, and usable package BUGS (Spiegelhalter *et al*, 1996) deserves special notice. This package has an intuitive language (as well as a graphical user interface) for expressing models, and does not require (or allow) the user to specify an algorithm with which to attack a problem. Our feeling is that if a problem can be handled by BUGS, then as of today one should normally use BUGS to solve it. However, we believe that researchers who develop new models will have an easier time extending YADAS's open source code than BUGS, so the effort spent learning YADAS will pay off.

MLwiN (Rasbash *et al*, 2000) is another package with many features including a graphical user interface and graphical diagnostics. This package supports classical analyses as well as MCMC. Most of their examples are hierarchical generalized linear models, admittedly a large and useful class of problems, but it does not appear to be straightforward for users to extend the capabilities of the package. MLwiN is not free.

Bassist (Toivonen *et al*, 1999) is another good package; unfortunately, it appears that it is no longer being supported. It converts code written using a model description language into C++ code. It has good extensibility since it can use C++ functions to define model parameters, but it is limited in its capability for generating different sorts of MCMC algorithms.

HYDRA (Warnes, 2002) is also written in Java and distributed open-source. It does not seem to have any limits to its extensibility, but neither does it provide classes that make it easy to specify new models (one has to define the posterior distribution from scratch) or to define new MCMC algorithms, although several special algorithms are included.

Flexible Bayesian Modeling (FBM) is a set of UNIX based tools for many sorts of Bayesian regression models. See Neal (2003).

Since YADAS is written in Java, it should work on most platforms, and we believe that the open source distribution and the versatility and extensibility of the model description and algorithm definition capabilities of YADAS ensure that it is the right choice for many users.

1.3 Structure of this paper

The paper is structured as follows. In §2, we introduce several problems of interest to Los Alamos researchers that are challenging to analyze. §3 contains an introduction to the software architecture and explains the power and flexibility of the design. In §4 we explain how this architecture makes it straightforward to implement these examples. §5 contains discussion, including comparison to existing systems, and ideas for future improvements.

2 Some problems tractable in YADAS

In this section we discuss several problems that are challenging to analyze without our system. First, we discuss estimating the prevalence of a binary characteristic in a population, where the prevalence varies across manufactured lots, and where the data include both random sampling and convenience sampling from the lots. In §2.2 we discuss a model for weekly mortality data, where weekly expected values follow peak behavior and where the severity, duration, and start of the seasons follow a hierarchical model. Next, in §2.3 we discuss the reliability modeling of systems of components, where one may have test data and/or expert opinion on components, subsystems, and the whole system. The framework is even capable of analyzing the results of auto races under a rich family of models for permutation data (§2.4). Finally, in §2.5 we present a model for software reliability that requires mixture distributions for unknown parameters as well as a new way of handling expert opinion. The point is not that all these models are exactly appropriate or that the problems could not have been solved acceptably without them, but that YADAS was able to facilitate analyzing them, and is likely to be helpful in analyzing many models we encounter in the future.

2.1 Random and convenience sampling from lots

One problem for which YADAS greatly accelerated obtaining a solution involved the estimation of the fraction p of items in a finite population that had a certain feature; see Booker *et al* (2003). The items were manufactured in lots, and it was believed that the presence or absence of the feature was related to lot membership: let p_i be the probability that an item manufactured in lot i would contain the feature, and assume that the p_i's come from a Beta(a, b) distribution, where a and b are unknown and have prior distributions. Items were sampled from some of the lots and inspected to determine whether they contained the feature. To complicate matters, the sampling proceeded in two phases, and while the second phase consisted of random samples within the lots, we were not willing to assume that the first phase of sampling generated items independently of the presence or absence of the feature (we will refer to this as the convenience sample). For this reason, we analyzed these samples using the *extended hypergeometric* distribution. Let N_i be the number of items in lot i, suppose that K_i of these items contain the feature, where K_i has a binomial distribution with parameters N_i and p_i, and suppose that the size of the ith convenience sample was n_i^C. Let θ be a parameter that measures the extent to which items with the feature are more likely to be sampled than items without ("bias"). Then the probability distribution for y_i^C, the number of items inspected that contain the feature, is

$$P(y_i^C = y) = \frac{\binom{n_i^C}{y}\binom{N_i - n_i^C}{K_i - y}\exp(\theta y)}{\sum_{j=\max(0, n_i^C - N_i + K_i)}^{\min(n_i^C, K_i)}\binom{n_i^C}{j}\binom{N_i - n_i^C}{K_i - j}\exp(\theta j)} \; ,$$

for $y = \max(0, n_i^C - N_i + K_i), \ldots, \min(n_i^C, K_i)$.

This reduces to the hypergeometric distribution when $\theta = 0$, and when $\theta > 0$, items with the feature are more likely to be sampled. We placed a $N(0, 1)$ prior on θ, indicating that our prior information did not favor biases in one direction or the other, but we wanted our uncertainty estimates to allow for the possibility that bias may exist. The extended hypergeometric distribution (Harkness, 1965) arises in the power function for Fisher's exact test for 2×2 tables, but to our knowledge this is the first time it has been used to handle possibly biased sampling. The interpretation of the model here is that if each item with the feature in lot i finds its way into the sample independently with probability π_{1i}, and each item without the feature appears in the sample independently with probability π_{0i}, then $\theta = \log(\pi_{1i}(1 - \pi_{0i})/\{\pi_{0i}(1 - \pi_{1i})\})$ is the same for each i. (The sampling was presumably conducted in a slightly different way, but this model should still be a useful way of parameterizing sampling bias.) Furthermore, after the convenience samples were taken, further random samples from the remaining items in each lot were taken (according to the hypergeometric distribution) and inspected for the presence of the feature.

The parameters of most interest are the K_i's, because from them, one can reconstruct the numbers of items that were not inspected but that contain the feature and hence estimate the prevalence of the feature in the uninspected part of the population. These parameters have discrete distributions, and the extended hypergeometric distribution was needed for inference, so writing code to estimate the model was challenging in principle. However, with YADAS, we were able to fit the model in less than an afternoon, rather than being forced to make inappropriate assumptions while using another model. See §4.1 for details on how we estimated the model using YADAS.

2.2 A hierarchical model for seasonal disease counts

Graves and Picard (2002) were presented with data which were counts of weekly mortality in Albuquerque hospitals from causes related to pneumonia and influenza (P&I), over a period of forty years. After exploratory analysis, they determined that the expected weekly mortality curve in a given season had approximately the shape of a Gaussian density. The authors wanted to allow the baseline, height, width, and location of these curves to vary according to a hierarchical model across years. If the random mortality for week t of year (season) s is denoted by $M_s(t)$, then they assumed

$$E\{M_s(t)\} = b_s + \frac{c_s}{\sigma_s}\phi\left(\frac{t-\Delta_s}{\sigma_s}\right), \tag{1}$$

where the hierarchical assumptions include $\log c_s \sim N(\mu_c, \sigma_c^2)$, etc., and μ_c and σ_c have prior distributions. Exploratory analysis also revealed that variability around this mean curve was larger than the Poisson distribution would predict, so the authors used an overdispersed model involving the negative binomial distribution (see McCullagh and Nelder, 1989). (The authors also estimated a model with another level of hierarchy, analyzing several cities simultaneously.)

This model has several interesting features: the regression function has an unusual form, the data distribution is negative binomial, and the first implementation of the model suffered from poor mixing due to correlations between, for example, the c_s's and μ_c. YADAS, then, needed to be extensible enough to handle new data distributions and new regression functions such as (1), and it also had to allow the users to experiment with techniques for improving the mixing of the MCMC algorithm. It passed all these tests; see §4.2 for details.

2.3 System reliability

A common problem dealt with by Los Alamos statisticians is the modeling of the reliability of multicomponent systems (see Johnson *et al.* (2003), Martz, Waller, and Fickas (1988), Martz and Waller (1990), Martz and Baggerly (1997)). An example system is shown in Figure 1; see also `yadas.lanl.gov`. Components 5, 6, and 7 are combined in series to form subsystem 2, components 3 and 4 combine in parallel to make subsystem 1, and finally subsystems 1 and 2 combine in series to make the full system, labeled 0. Suppose the components and subsystems are labeled from 0 to $n-1$. The probability that component or subsystem i works properly is defined to be p_i. The structure of the system defines functional relationships between the p_i's; for example, $p_2 = \prod_{i=5}^{7} p_i$, and $p_1 = 1-(1-p_3)(1-p_4)$. In other words, the only independently varying p_i's are those represented by leaf nodes in pictures such as Figure 1, in which nodes 3 through 7 are leaf nodes. We may have test data in the form of x_i successes in n_i trials for component or subsystem i. We may also have one or more experts providing their input on the reliabilities of the components and subsystems: we assumed that expert j's beliefs take the form of point estimates $\tilde{p}_{ij}$ for the reliability of component i, and strengths of beliefs ν_{ij}. We translate these prior beliefs as if they were binomial data with potentially noninteger numbers of successes. The p_i's for leaf nodes are in addition given prior distributions (sometimes hierarchically, if we believe *a priori* that some subsets of components have approximately the same success probabilities). We then place prior distributions on the ν_{ij}'s. This gives rise to a posterior distribution proportional to

$$\prod_{i=0}^{n-1}\left\{p_i^{x_i}(1-p_i)^{n_i-x_i}\prod_j p_i^{\nu_{ij}\tilde{p}_{ij}}(1-p_i)^{\nu_{ij}(1-\tilde{p}_{ij})}\right\} \times \text{prior}(p,\nu).$$

Again, the p_i's are not allowed to vary independently because there are functional relationships between them, so that posterior distributions are not simply beta with parameters that can be read from the expression above. We seek posterior distributions of all the p_i as well as the ν's. Using YADAS, we created a system for studying general systems of this form. We explain how in future sections.

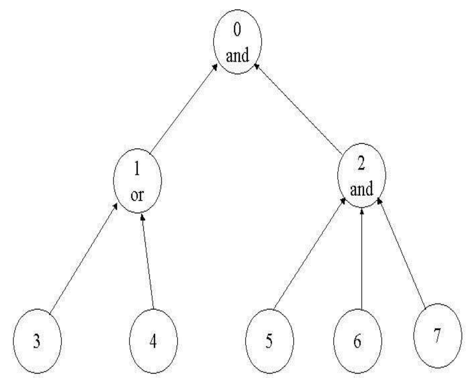

Figure 1: Diagram of an Example System. The success probability of the system (node 0) is $\{1-(1-p_3)(1-p_4)\}p_5p_6p_7$.

2.4 Auto racing results

A dirty, poorly kept secret about YADAS was that most of its key ideas emerged during a project to model stock car racing results and hence to celebrate the performance of the author's cousin. Full discussion of this analysis is given in Graves, Reese, and Fitzgerald (2003). To analyze auto racing results, one needs a family of models for permutations that depend on ability parameters for each driver. Then one needs to place additional structure on the ability parameters in order to estimate interesting interactions between drivers and various factors. The basic model is as follows. Suppose n drivers participate in a race. Let the ability of the ith driver be denoted by θ_i, where the θ_i's have an independent mean zero normal distribution with unknown variance. The probabilistic model determining finishing order conditional on ability parameters works by choosing the last place finisher with probability proportional to $\lambda_i = \exp(-\theta_i)$, choosing the second-to-last finisher among the remaining drivers with probabilities still proportional to λ_i, and continuing until two drivers i_1 and i_2 have been determined to finish first and second; driver i_1 wins with probability $\lambda_{i_2}/(\lambda_{i_1}+\lambda_{i_2})$. To analyze many races simultaneously, we assume that driver i has ability θ_{ij} in race j, and let $\lambda_{ij} = \exp(-\theta_{ij})$ Let π_j be defined so that $\pi_j(k) = i$ means that driver i finished in kth place in race j; then the likelihood function is

$$P(\pi|\lambda) = \prod_{j=1}^{J}\prod_{i=1}^{I_j} \lambda_{\pi_j(i),j}\left(\sum_{k=1}^{i}\lambda_{\pi_j(k),j}\right)^{-1}.$$

This is clearly a novel likelihood function. Interesting ways of parameterizing θ_{ij} include $\theta_{ij} = \theta_i$ (driver abilities are constant over all races) and assuming that driver abilities are changing linearly in time. One may also study the extent to which some tracks are more predictable than others (i.e., $\theta_{ij} = \theta_i\phi_{T(j)}$, with large values of the positive parameter ϕ_t implying that results at track t are highly predictable). We also studied the possibility that there was a track-driver interaction: each track-driver combination (i,t) leads to a different parameter θ_{it}, with the θ_{it}'s distributed around an overall driver ability θ_i. One may also propose regression relationships based on track properties like speed, length, and amount of banking.

We did not find any indication that existing packages would help us analyze this family of models, so we created our own code, and Java encouraged us to overengineer a very versatile solution.

2.5 Software reliability with mixture distributions and a novel way of handling expert opinion

In a software testing and reliability scenario (see Kern, Graves, and Ovalle 2003), suppose that each possible test t of the software has probability $p(t)$ of yielding correct behavior. This formulation can make sense even with deterministic software, if t is in fact only a partial description of the test, so that many tests share the same description and may be handled differently by the software. However, we want to allow the case where $p(t) = 1$. Suppose further that a list of requirements for the software has been converted into a list of tasks, and a given test can either exercise or not exercise each of these tasks; let x_{tj} be the indicator that test t exercises task j. We then assume that the failure probability satisfies

$$\mathrm{logit}\{1 - p(t)\} = \sum_j \beta_j x_{tj}.$$

Here $\beta_j \geq 0$. In fact, we assume that the prior distribution of β_j is the gamma-point mass distribution, a mixture of a point mass (of probability p_0) at zero and a gamma distribution with shape parameter α and scale parameter s. $p_0, \alpha,$ and s are each given prior distributions.

To complicate matters further, we also assume that expert opinion on the difficulty of each testing task was available. We wish to incorporate this opinion in the manner of Graves (2003), which we discuss here in the continuous case although that is not the relevant case for this example. In an ordinary hierarchical model, one assumes $\beta_j \sim G$, which is equivalent to assuming that $G(\beta_j)$ has a uniform $(0, 1)$ distribution. We generalize this to an assumption that $G(\beta_j)$ has a beta distribution with parameters $a_j + 1$ and $b_j + 1$; in particular, we translate the expert opinion into mean values $\pi_j = a_j/(a_j + b_j)$ and let $a_j + b_j = M$ be a random constant independent of j that measures the trustworthiness of the expert. $M = 0$ is the usual hierarchical model assumption. This approach is straightforward to insert into analyses, because if we let B_j denote the beta distribution function with parameters a_j and b_j, and g the continuous density function for G then the posterior distribution is modified by replacing $g(\beta_j)$ by $g(\beta_j)B_j(G(\beta_j))$.

This model inspires some new difficulties for MCMC computation. First, it has some novel terms in the posterior distribution arising from the gamma-point mass distribution augmented by expert opinion. Also important is that the β_j's have mixture distributions, requiring the algorithm to sample from them. Reversible jump MCMC (Green, 1995), in which the Markov chain jumps between two or more different spaces in which a parameter(s) can live, helped us tackle this problem: in fact YADAS features some fairly general reversible jump capability.

3 The architecture of YADAS: a taxonomy of (Bayesian) statistical models and MCMC algorithms

The software architecture of YADAS is actually quite simple, as well as being intuitively appealing. The basic idea in our system for expressing models is that posterior distributions are products of terms; we call these *bonds* and have a useful set of components for expressing them. (The term 'bond' comes from a chemical interpretation of a graphical model in which parameters are atoms.) In this section we provide some introduction to object-oriented concepts, since YADAS is written in Java and since YADAS would not have been imagined without object-oriented ideas. Afterwards we introduce the basic classes in YADAS: the `MCMCParameter`, the `MCMCBond`, the `ArgumentMaker`, the `Likelihood`, and the `MCMCUpdate`. In a typical analysis, one begins by defining a collection of `MCMCParameters`. Then one lists `MCMCBonds`, which are relationships between these parameters. Next one defines a collection of `MCMCUpdates`, which are ways in which the algorithm will try to update the parameters. One loops over the updates, and writes values of parameters to files named after the parameters. Within an update, a new value for one of more parameters is proposed, and the software loops over and computes all the relevant bonds in order to help determine the

Metropolis acceptance probability for the move, which is then either accepted or rejected.

3.1 Some object-oriented terminology

Object-oriented programming consists of manipulations of *objects*. The abstract description of like objects is referred to as a *class*; a specific object used in a program that is an example of a class is known as an *instance* of that class. (A class is a sort of template, and instances are things that follow that template.) Classes are like data structures in non-object-oriented programming in that they can contain several pieces of data of different forms, but they also can contain *methods*, as functions are known. Object-oriented systems also have class hierarchies: if a class is built based on an existing class to contain all the same functionality (perhaps with some of it reimplemented) perhaps along with some additional functionality, it is known as a *subclass* of the first class or is said to *inherit* from that class; one often sees extensive diagrams of inheritance structure for large systems. Another important concept is the *interface*: an interface is a collection of names of methods, and a class is said to *implement* the interface if it contains definitions for all those methods. In Java, a class may inherit from only one superclass, but it may implement arbitrarily many interfaces. The interface leads to one of the most important object-oriented programming techniques: one can construct an array or other collection of objects that implement the same interface, and then loop over them, calling methods that belong to the interface, which can lead to somewhat different appropriate behavior for each object. In what follows we discuss the three main pieces of a YADAS analysis: instances of the `MCMCParameter` class, objects implementing the `MCMCBond` interface (almost always instances of `BasicMCMCBond`, which in turn uses two important concepts, the `ArgumentMaker` and the poorly named `Likelihood`), and the `MCMCUpdate` interface.

Many good introductory books on Java and object-oriented programming exist; a favorite is Eckel (1998).

3.2 `MCMCParameters`

The first basic element in the YADAS architecture is the `MCMCParameter`. All quantities whose posterior distribution the analysis seeks and hence which get updated in the algorithm are parameters. `MCMCParameters` can be multivariate if desired; for instance in an analysis of variance problem with group-dependent means, the entire mean vector can be stored in the same parameter. Also entire vectors of regression parameters are typically stored together. There is some freedom in how parameters are defined: for example, data that are not updated can be stored in parameters.

In the software, a parameter consists of a vector of values that are to be updated, a vector of step sizes for Metropolis steps, and a name for a file to which to write output.

A critical piece of functionality in the `MCMCParameter` class is the ability of each `MCMCParameter` to update itself using componentwise Metropolis steps. We will discuss the `MCMCUpdate` interface soon, but it is most important to understand that `MCMCParameter` is one class that implements the `MCMCUpdate` interface. When the `update()` method of a `MCMCParameter` is called, we loop over each component in the parameter. For each component, we construct a proposed new value of that component by adding its step size multiplied by a standard normal variable. We then compute the ratio of the posterior for the new value relative to the old value by looping over bonds containing the parameter. The new value is then accepted if a uniform random variable is less than the posterior ratio.

There are some subclasses of the `MCMCParameter` class: most simply, the `MultiplicativeMCMCParameter` and `LogitMCMCParameter` classes are identical to the basic `MCMCParameter` class, but they update themselves by making Metropolis-Hastings proposals on the log scale (for positive valued parameters) and logit scale (for parameters with support in $(0, 1)$), respectively.

3.3 Bonds

The `MCMCParameter` class in YADAS is fairly straightforward. The `MCMCBond` interface is more critical for making YADAS usable. Its responsibility is making it possible to express any possible term in a prior distribution or a likelihood function. The critical functionality in the Bond interface is the ability to compute the value of the bond for new values of the parameters. A simple example of a bond is a Gaussian relationship between three parameters: an n-vector of data y, an n-vector of means μ, and an n-vector of standard deviations σ. Computing this bond means evaluating

$$\sum_{i=1}^{n}\{-\log\sigma_i - (y_i-\mu_i)^2/2\sigma_i^2\},$$

since we do computations on the log scale.

Most bonds that we work with belong to the `BasicMCMCBond` class, which is very versatile, and which saves much of the coding work. It is based on the observation that there are only a few different commonly used probability density functions: `Gaussian`, `Gamma`, `Poisson`, `Binomial`, `Beta`, t, and so forth, and multivariate versions of some of these. The richness of the family of potential data analyses comes from the variety of ways of selecting arguments for these density functions. A `BasicMCMCBond`, then, consists of a vector of parameters, a vector of `ArgumentMaker`s, which are essentially functions that convert parameters into arguments for a density function, and a `Likelihood`, which takes the output of the `ArgumentMaker`s, runs it through the density function, and returns the scalar value of the bond.

3.3.1 `ArgumentMakers`

`ArgumentMaker` is an interface with the single method `getArgument`; this method takes a two-dimensional array of parameter values and returns a one-dimensional array of outputs to be sent to the likelihood function. Here is a partial list of the `ArgumentMaker`s that we have found useful thus far.

- `ConstantArgument`. This class ignores the values of the parameters and always returns a predetermined array of data, sometimes many copies of the same value, sometimes an array of different data points.
- `IdentityArgument`. This class returns one of the parameters unchanged.
- `GroupArgument`. This class is useful when, for example, many data points share the same mean. It 'expands' a short vector of parameters into a larger vector. For example, if $y_i \sim N(\mu, \sigma^2)$ for $1 \le i \le n$, the μ and σ parameters have length one, and two `GroupArgument`s are needed, the first to create a vector of length n, all of whose elements are equal to the scalar μ, and another to do the same for σ.
- `FunctionalArgument`. `FunctionalArgument` is very powerful: its purpose is to run one or multiple parameters through an arbitrary function specified by the user, possibly after 'expanding' them first as in `GroupArgument`. A simple example where we used a functional argument was in a case where normal data had standard deviations proportional to their means: in this case a `FunctionalArgument` expanded a scale parameter to the length of the data, then took the absolute value of the product of this scale parameter and the means.
- `LinearModelArgument`. The class `LinearModelArgument` defines linear models, including those with both numeric and categorical variables. It also contains the capability for running the linear predictor through an arbitrary (link) function. These arguments can then be used as the mean or the standard deviation in a normal linear model, or equally in a generalized linear model.

3.3.2 `Likelihoods`

`Likelihoods` in YADAS are simply functions like the normal example above; they are sent vectors of arguments in standard orders and compute quantities such as $\sum_{i=1}^{n}\{-\log\sigma_i - (y_i-\mu_i)^2/2\sigma_i^2\}$, or the obvious analogues for other familiar density functions. The `Likelihood` interface would perhaps be better named `LogDensity`. Note that there are some symptoms of computational inefficiency here; we compute terms like the $-\log\sigma_i$ term even if σ_i is not being updated, in which case most Bayesian statisticians would think of this term as a constant.

3.4 Updates

YADAS also contains an Update interface, so that it is possible to make MCMC moves of completely different forms than the standard componentwise Metropolis update. With this construct, YADAS is an environment for conducting research into novel ways of updating parameters to improve MCMC convergence. In particular, we hope that researchers will good Gibbs sampling-based ideas will find it worthwhile to experiment with Metropolis–Hastings approximations of these ideas using YADAS.

The Update interface is based on a single method, `update()`. As mentioned before, the `MCMCParameter` class implements the update interface, and in many analyses, the only updates are the parameters. (To reiterate, the way that `update()` is implemented in the `MCMCParameter` class involves looping over components, and attempting to add a Gaussian amount with a specified step size to each in turn.)

However, the update interface is more general. If one wanted to do pure Gibbs sampling in YADAS, it would be possible because one could write routines that sampled from full conditional distributions for the model at hand and place those routines inside classes implementing `Update` (this would generally be a waste of the YADAS architecture, however). We prefer instead to use Metropolis(-Hastings) updates in which multiple parameters and/or multiple components of individual parameters are updated simultaneously. This belief is based on the idea of having highly correlated parameters that move sluggishly when updated individually. To facilitate writing these sorts of update, we wrote a `MultipleParameterUpdate` class that reduces the task of writing a complex update to writing a `Perturber`. `Perturber`s are functions that take current values of several parameters as input, and generate new values of those parameters as a candidate. `Perturber`s need to calculate density ratios for their proposal distributions for correct Metropolis-Hastings acceptance probability calculations. Examples of `Perturber`s that we have used include adding a common constant to each component of a vector parameter and to the scalar parameter that serves as the mean of the vector parameter. We have also done similar things modifying the scale of entire parameters rather than their center. In regression problems where it is not possible to center the data, one may wish to add a constant to the intercept α while simultaneously subtracting a related constant from the slope β in order to keep $\alpha + \beta\bar{x}$ constant, where $\bar{x}$ is the average of the covariates.

Another class of updates is necessary when an updatable parameter has discrete support. If this support is finite, YADAS will typically handle this case using a Gibbs step, by computing the posterior ratio for each possible move of the parameter, and selecting the next value of parameter proportionally to these ratios. See §4.1 for an example.

We have also provided support for reversible jump MCMC (Green, 1995). Reversible jump MCMC can be used when a parameter's support is best thought of as a union of two or more spaces. The `ReversibleJumpUpdate` class is used here: conceptually, this class requires a matrix of transition probabilities for proposing moves between the spaces, and a matrix of proposed moves for each ordered pair of spaces. The user also needs to supply transition density functions for these moves, so that use of this class is not as automatic as in other classes. We have used the `ReversibleJumpUpdate` class on the software reliability problem discussed earlier.

3.5 Summary of YADAS Architecture

After all the parameters, bonds, and updates in an application are defined, the remainder of the code in a YADAS application is simple. The algorithm consists of looping over the array of `MCMCUpdate`s, attempting each update in each iteration. Most of these updates in turn loop over the bonds in order to calculate ratios of values of the posterior distribution. After the last update of an iteration has been attempted, one sends the current values of the parameters to output files.

We claim that the `MCMCBond` interface (most often using `BasicMCMCBond`s) and the `Update` interface are both sufficient to handle all Bayesian statistical models, though this is difficult to make mathematically precise. It is a tautology that all models are products of one or more bonds, and that all MCMC algorithms can be expressed as sequences of updates, but this observation provides no insight into the usefulness of the YADAS architecture. What is important is whether the YADAS architecture provides an easy way to analyze models composed of standard components, and whether it is extensible to handle a large class of problems not yet implemented using it.

4 How to implement the examples in YADAS

In this section we illustrate the versatility of YADAS using the examples discussed earlier. In many cases, most of what is required of an analyst when implementing an entirely new model is writing the code for one or more new `ArgumentMaker`s, which is not particularly challenging, since these are simply functions that take two-dimensional arrays as input and return one-dimensional arrays, and figuring out how to code them does not require any complicated calculations since it is just expressing the model. Occasionally one will need to code a new `Likelihood` function, and advanced users may want to experiment with `Perturber`s as part of `MultipleParameterUpdate`s.

4.1 Sampling from lots

In §2.1 we discussed a hierarchical model for a collection of probabilities, estimated using data with hypergeometric and extended hypergeometric distributions. To analyze these data using YADAS, we needed only to write the `ExtendedHypergeometric` probability density function. This is a class that implements the `Likelihood` interface, so that it defines a `lik()` method that when called with arguments y, n, N, K, and θ, it returns

$$\sum_i \log\left\{\frac{\begin{pmatrix} n_i \\ y_i \end{pmatrix}\begin{pmatrix} N_i - n_i \\ K_i - y_i \end{pmatrix}\exp(\theta_i y_i)}{\sum_{j=\max(0,n_i-N_i+K_i)}^{\min(n_i,K_i)}\begin{pmatrix} n_i \\ j \end{pmatrix}\begin{pmatrix} N_i - n_i \\ K_i - j \end{pmatrix}\exp(\theta_i j)}\right\} .$$

In addition, we wrote a standard hypergeometric (i.e. $\theta_i \equiv 0$) density function at this point for increased computation speed, although this would not have been necessary.

Note that the `ArgumentMaker` construct made it easier to use the hypergeometric code. Recall that the randomly sampled (hypergeometric) data were obtained from what remained after the (extended hypergeometric) convenience sample was removed. In other words, the random data y_i^R had a hypergeometric distribution with sample size n_i^R, population size $N_i - n_i^C$, and with $K_i - y_i^C$ items in the population containing the feature. The K_i's are unknown and hence their values change throughout the algorithm. A `FunctionalArgument` that computed the $(K_i - y_i^C)$'s enabled us to use the ordinary hypergeometric code without writing a version suitable for two phases of sampling.

Finally, the parameters of most interest, the K_i, have discrete distributions, so the MCMC algorithm needed to sample them appropriately. The `FiniteUpdate` class, which implements the `MCMCUpdate` interface, samples from the complete conditional distributions of parameters whose support is $\{0, 1, \ldots, S-1\}$. Note that YADAS is conveniently set up to do this. Its architecture, based on the Metropolis algorithm, makes it natural to compute terms like $r_i = f(i)/f(s_0)$, where $f(\cdot)$ denotes the unnormalized posterior evaluated at the value of the discrete parameter and where s_0 is the current value. Sampling the new value of the discrete parameter with probabilities proportional to the r_i is then exactly a Gibbs step for this parameter, if one can stomach the repeated redundant computation of $f(s_0)$. In some cases, it is wasteful to consider all possible values of the discrete parameter, in which case the `IntegerMCMCParameter` class can be used instead. It implements a Metropolis step where the proposed value of the discrete parameter is a discretized Gaussian step from the current value.

4.2 Weekly mortality data

In §2.2 we discussed the hierarchical modeling of weekly P&I mortality data, where the weekly means were assumed to follow a curve shaped like the Gaussian density, and where the data distribution was an overdispersed Poisson, or negative binomial. To perform this analysis, it was necessary to write a `FluArgument` class, which was capable of taking as input the current values of the parameters b, σ, Δ, and c, and return an array whose length is the number of data points and whose ith entry is

$$\mu_i = b_{s(i)} + \frac{c_{s(i)}}{\sigma_{s(i)}} \phi \left(\frac{t(i) - \Delta_{s(i)}}{\sigma_{s(i)}} \right),$$

where $s(i)$ indicates from which season the ith data point came, $t(i)$ indicates to which week the ith data point corresponds, and $\phi(x) = (2\pi)^{-1/2} \exp(-x^2/2)$. Also, we needed the `OverdispersedPoisson` density function, which when called with arguments y, μ, and ψ, returns

$$\sum_i \log \left\{ \frac{\Gamma(y_i + \psi_i \mu_i) \psi_i^{\psi_i \mu_i}}{y_i! \Gamma(\psi_i \mu_i)(1 + \psi_i)^{y_i + \psi_i \mu_i}} \right\}.$$

With these two pieces in place, YADAS code to specify the likelihood function for these data is:

```
bondlist.add ( databond = new BasicMCMCBond
    ( new MCMCParameter[] {b, sigma, delta, c, psi},
      new ArgumentMaker[]
          { new ConstantArgument (d.r("y")),
            new FluArgument (d.i("year"), d.r("week")),
            new GroupArgument (4, d.i(0)) },
      new OverdispersedPoisson() ));
```

As always, the definition of the `BasicMCMCBond` begins with a list of the parameters involved in the bond, continues with the list of argument functions, and finishes with the density function name. Here the arguments are the data y, the mean (`d.i(‘‘year’’)` and `d.i(‘‘week’’)` help to specify the functions $s(\cdot)$ and $t(\cdot)$ defined in the description of `FluArgument` above), and the precision parameter of the negative binomial distribution: `GroupArgument (4, d.i(0))` makes as many copies of the fourth parameter (ψ, since indexing starts at zero) as there are data points.

Finally, the parameters b, σ, Δ, and c with hierarchical priors were prone to poor mixing since (for example) all the Δ_i were correlated with each other and with their prior mean parameter μ_Δ. We alleviated these mixing problems with four special update steps involving `AddCommonPerturbers`. These updates add Metropolis steps to the algorithm in which the proposed new values of the Δ_is are $\Delta_i + \epsilon Z$ and in which the proposed new value of μ_Δ is $\mu_\Delta + \epsilon Z$. Here $Z \sim N(0, 1)$, the same Z is used for each i, and $\epsilon > 0$

is a tunable step size. These proposals move the parameters in a direction of relatively high posterior variability; functions of Δ orthogonal to this direction have low variance due to the presence of the term $-\frac{1}{2\sigma_\Delta^2}\sum_i(\Delta_i-\mu_\Delta)^2$ in the posterior. We implemented similar update steps for b, σ, and c, on the log scale where appropriate.

4.3 System reliability

In §2.3 we discussed an important model, presented in Johnson *et al* (2003), for estimating the reliability of a complex system, with several sources of data and expert opinion. The primary issue in defining this model in YADAS is telling it how to compute the success probabilities of the (sub)system nodes as a function of the success probabilities for the leaf nodes. (We define a parameter vector `p` that contains only the leaf node probabilities.) To help with these problems we created a `ReliableSystem` class. When constructed it consists only of a collection of empty nodes. However, its structure can be usefully defined by calling its `SPIntegratorsFromFile` method is called as in

```
system.SPIntegratorsFromFile (d.i("parents"), d.i("gate"));
```

here `system` is an instance of the class `ReliableSystem`. The `parents` array indicates the parent of each node: if a node is part of a subsystem higher up the tree, that subsystem is its parent; for example, node 2 is the parent of nodes 5, 6, and 7 in Figure 1. The `gate` array indicates whether the subsystem defined at each node is constructed by combining its children in series or in parallel. After this, one can incorporate binomial testing data using

```
bondlist.add( binomialdata = new BasicMCMCBond
    ( new MCMCParameter[] { p },
      new ArgumentMaker[] { new ConstantArgument (d.r("x")),
                            new ConstantArgument (d.r("n")),
                            system.fillProbs (d.i("pexpand"),
                                              d.i("order")) },
      new Binomial () ));
```

Here the data are x_i successes in n_i trials, and the `system.fillProbs()` method evaluates the appropriate success probabilities p_i using the system structure. (`pexpand` maps the unknown parameter p to the leaf nodes in the graph, while `order` tells the method the order in which the probabilities should be computed.) We also use the `system.fillProbs()` method to incorporate expert opinion with beta distributions.

Sampling from this model has been relatively easy in our experience, but one important technique for high- (or low-) reliability systems is updating the unknown probabilities on the logit scale. The class `LogitMCMCParameter` handles all the Metropolis-Hastings details. The YADAS implementation of this model has successfully been compared to the results of an independent C implementation.

4.4 Auto racing results

The auto racing model has an unusual likelihood function, but analyzing the model is still not particularly difficult. The special class used in the analysis is `AttritionLikelihood`, which stores a vector of race numbers and a vector of finish positions, and takes two arguments corresponding to a vector of θ's parameterizing driver abilities (these can depend on properties of the race) and a vector of ϕ's parameterizing the predictability of the races (most often, these ϕ's are identically one). `AttritionLikelihood` then computes the logged probability of the competitors finishing in the order they did, conditional on the unknown parameters: recall that the model calls for the kth place finisher in the jth race to be chosen from the top k finishers with probability proportional to $\exp(-\theta_{ij}\phi_j)$ The θ and ϕ vectors can easily be constructed using `GroupArgument`s. The great benefit of this structure was that only one `AttritionLikelihood` function was needed, and several models (driver abilities, track predictabilities, trends in ability over time, track-driver interactions) could all be studied using the same `Likelihood` with different `ArgumentMaker`s.

Some special updates have been helpful in our analyses of auto racing results. For example, adding a common constant to the θ's has no effect on the likelihood, and perhaps for this reason, the average value of the θ's can wander slowly in our algorithms, with the individual θ's maintaining relatively constant distances from the average. This is not really a serious issue, but we sometimes add a `MultipleParameterUpdate` adding a common constant to the θ's, and this superficially improves the look of the iterations. More serious is the fact that the overall scale of the θs can change slowly. If the prior distribution for the θ's is independent $N(0, b^2)$, and b has a gamma hyperprior distribution, then we observe that b and the sample standard deviation of the θ's are highly correlated. We obtain an algorithm with better mixing with respect to b by adding a `MultipleParameterUpdate` where the proposed move is

$$\begin{aligned}
\theta_i &\leftarrow \bar{\theta} + c(\theta_i - \bar{\theta}) \ (1 \le i \le I), \quad \text{where } \bar{\theta} = I^{-1}\sum_{i=1}^{I}\theta_i; \\
b &\leftarrow cb,
\end{aligned}$$

with c chosen lognormally. Beyond that, if the ϕ parameters are also estimated, we find that the sample standard deviation of the θ's is highly negatively correlated with the geometric mean of the ϕ's, because the scale of the θ's could be multiplied by c while all the ϕ's are divided by c, without changing the likelihood. Suppose the prior for the ϕ's is exponential with mean d. We add a proposed Metropolis-Hastings move of

$$\begin{aligned}
\theta_i &\leftarrow \bar{\theta} + c(\theta_i - \bar{\theta}) \ (1 \le i \le I), \ \text{where } \bar{\theta} = I^{-1}\sum_{i=1}^{I}\theta_i; \\
b &\leftarrow cb; \\
\phi_j &\leftarrow \phi_j/c \ (1 \le j \le J); \\
d &\leftarrow d/c,
\end{aligned}$$

again with c chosen lognormally.

4.5 Software reliability

Our final example deals with a logistic regression model for software testing, where the regression coefficients β_j had a distribution which was a mixture of a point mass at zero and a gamma distribution, and further, we used the probability integral transform to incorporate expert opinion; see §2.5 and Kern *et al* (2003). We had to write the `ExpertGammaPointMass Likelihood`; we have already discussed how to write new density functions. Of more interest is the MCMC algorithm, where we use reversible jump MCMC (Green, 1995) to move between $\beta_j = 0$ and $\beta_j > 0$. YADAS features fairly general reversible jump MCMC support, although many specific examples will retain challenges even if one uses YADAS. To define a reversible jump algorithm in YADAS, one specifies a matrix (A_{ij}) of transition probabilities between states (in this example, the two states are $\beta_j = 0$ and $\beta_j > 0$). Also, one defines as many instances of the interface `JumpPerturber` as there are entries in the transition matrix. A `JumpPerturber` consists of a function that generates proposed new values of the parameters, and a function that computes the probability density $T_{ij}(\beta, \beta')$of the randomly generated new parameter value. In the case that the current value of the parameter is β, which is in state i, and the proposed new value is β', which is in state j, YADAS computes the acceptance probability

$$\frac{\pi(\beta')}{\pi(\beta)}\frac{A_{ji}}{A_{ij}}\frac{T_{ji}(\beta', \beta)}{T_{ij}(\beta, \beta')},$$

where π is the unnormalized posterior distribution written as a function of β alone. As usual in the Metropolis-Hastings rule, we then accept the new value β' or remain at β. YADAS's reversible jump capability is general enough to propose simultaneous moves of any subset of the parameters.

In the software testing example, we implemented an independent version of the code in C and obtained the same results as in YADAS.

4.6 Summary of examples

These examples have illustrated that YADAS is useful for analysis of a wide variety of new models. New likelihoods and prior structures are both reasonably easy to include in analyses, trivial in the common case that the building blocks already exist. Parameters can be updated in several ways with minimal effort, and reversible jump is even practical. Finally, YADAS encourages updates of multiple parameters at the same time, and this can be vital in obtaining tolerable mixing. These examples demonstrate that YADAS is likely to be relatively easy to adapt to many models not yet dreamed of.

5 Discussion

In this final section, we discuss what we hope to accomplish in the future. This is particularly important since we intend YADAS to be an environment that facilitates extensions and further research.

A vital direction of ongoing work on YADAS is the development of a more intuitive interface through an intermediate language such as that used by BUGS. Though we made an effort to make the process of writing Java code to develop YADAS applications as natural as possible, YADAS code still looks forbidding and scares away potential users. The challenging issue here is to develop an interface that is as extensible as the system itself: we do not want to have to update the interface code every time we add a new likelihood, argument function, or update method. We also hope to work within the Omegahat project (`www.omegahat.org`) to facilitate working simultaneously with YADAS and R (`www.R-project.org`): this would have the benefits of making it easier to construct input files for YADAS, and especially for making it more convenient to explore MCMC output to evaluate convergence and mixing.

We are also working toward the goal of making YADAS a preferred environment for research into methodology for improving mixing of MCMC algorithms. The `MultipleParameterUpdate` construct has had some impressive successes, and we are working on organizing the ideas (see Graves, Speckman, and Sun, 2003). In fact, we believe that ultimately YADAS will be capable of designing suitable updates of multiple parameters automatically. We have also begun providing support for parameter expanded data augmentation (PX-DA; see Liu and Wu (1999) and Meng and van Dyk (1999)) in some generality.

References

Berger, J.O. (2000), "Bayesian Analysis: A Look at Today and Thoughts of Tomorrow," *Journal of the American Statistical Assocation* 95:1269-1276.

Booker, J.M., Bowyer, C., Chilcoat, K., DeCroix, M., Graves, T.L., and Hamada, M.S. (2003), "Inference for a Population Proportion Using Stratified Data Arising From Both Convenience and Random Samples," in preparation.

Chib, S. and Greenberg, E. (1995), "Understanding the Metropolis–Hastings Algorithm," *The American Statistician* 49:327-335.

Eckel, B. (1998), *Thinking in Java*. Upper Saddle River, NJ: Prentice Hall.

Gelfand, A.E. (2000), "Gibbs Sampling," *Journal of the American Statistical Assocation* 95:1300-1308.

Graves, T.L. and Picard, R.R. (2002), "Seasonal Evolution of Influenza-Related Mortality," Los Alamos National Laboratory Report LA-UR-03-1237.

Graves, Reese, C.S., and Fitzgerald, M. (2003), "Hierarchical Models for Permutations: Analysis of Auto Racing Results," *Journal of the American Statistical Association*, 98:282-291.

Graves, T.L., Speckman, P., and Sun, D. (2003), "Characterizing and Eradicating Autocorrelation in MCMC Algorithms for Linear Models and More" In preparation.

Green, P.J. (1995), "Reversible Jump MCMC Computation and Bayesian Model Determination," *Biometrika* 82:711-732.

Harkness, W.L. (1965), "Properties of the Extended Hypergeometric Distribution," *Annals of Mathematical Statistics* 36:938-945.

Hastings, W.K. (1970), "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika* 57:97-109.

Johnson, V.E., Graves, T.L., Hamada, M.S., and Reese, C.S. (2002), "A Hierarchical Model for Estimating the Reliability of Complex Systems," *7th Valencia International Meeting on Bayesian Statistics*, Tenerife, Spain.

Kern, J.C., Graves, T.L., and Ovalle, N. (2003), "Hierarchical mixture models for software testing". In preparation.

Liu, J.S. and Wu, Y.N. (1999), "Parameter Expansion for Data Augmentation," *Journal of the American Statistical Association* 94:1264-1274.

Martz, H.F. and Baggerly, K.A. (1997), "Bayesian Reliability Analysis of High-Reliability Systems of Binomial and Poisson Subsystems," *International Journal of Reliability, Quality and Safety Engineering*, 4:283-307.

Martz, H.F. and Waller, R.A. (1990), "Bayesian Reliability Analysis of Complex Series/Parallel Systems of Binomial Subsystems and Components," *Technometrics*, 32:407-416.

Martz, H.F., Waller, R.A. and Fickas, E.T. (1988), "Bayesian Reliability Analysis of Series Systems of Binomial Subsystems and Components," *Technometrics*, 30:143-154.

McCullagh, P. and Nelder, J. A. (1989). *Generalized Linear Models*. Chapman Hall: London.

Meng, X.L. and van Dyk, D.A. (1999), "Seeking Efficient Data Augmentation Schemes via Conditional and Marginal Augmentation," *Biometrika* 86:301-320.

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E. (1953), "Equations of State Calculations by Fast Computing Machines," *Journal of Chemical Physics* 21:1087-1092.

Neal, R. (2003), "Software for Flexible Bayesian Modeling and Markov Chain Sampling"; see `http://www.cs.toronto.edu/~radford/fbm.software.html`.

Rasbash, J., Browne, W., Goldstein, H., Yang, M., Plewis, I., Healy, M., Woodhouse, G., Draper, D., Langford, I., and Lewis, T. (2000), *A User's Guide to MLwiN (Second Edition)*, London, Institute of Education.

Spiegelhalter, D.J., Thomas, A., Best, N.G., and Gilks, W.R. (1996), *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5 (version ii)*; see `http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml`.

Tierney, L. (1994), "Markov Chains for Exploring Posterior Distributions," *Annals of Statistics* 22:1701-1762.

Toivonen, H., Mannila, H., Seppanen, J., and Vasko, K. (1999), *Bassist User's Guide For Version 0.8.3*; see `http://www.cs.helsinki.fi/research/fdk/bassist/`.

Warnes, G.R. (2002), "HYDRA: a Java Library for Markov Chain Monte Carlo", *Journal of Statistical Software*, Volume 7 Issue 04, 03/10/02.